Chapter 13
Link Analysis

13.1 Search Engine

A search engine is an application which takes as input a search query and returns a list of relevant Webpages. Reference [3] explains in detail the general architecture of a typical search engine as shown in Fig. 13.1.

Every engine requires a *crawler* module. A crawler is a small program that browses the Web by following links. The crawlers are given a starting set of pages. They then retrieve the URLs appearing in these pages and give it to the *crawl control* module. The crawl control module determines what links to visit next, and feeds the links to visit back to the crawler. The crawler also stores the retrieved pages in the *page repository*.

The *indexer* module extracts all the words from each page, and records the URL where each word occurred. The result is a very large "lookup table" that has URLs mapped to the pages where a given word occurs. This index is the *text index*. The indexer module could also generate a *structure index*, which reflects the links between pages. The *collection analysis module* is responsible for creating other kinds of indexes, such as the *utility index*. Utility indexes may provide access to pages of a given length, pages of a certain "importance", or pages with some number of images in them. The text and structure indexes may be used when creating utility indexes. The crawl controller can change the crawling operation based on these indexes.

The *query engine* module receives and files search requests from users. For the queries received by the query engine, the results may be too large to be directly displayed to the user. The *ranking* module is therefore tasked with the responsibility to sort the results.

K. Raj P. M. et al., *Practical Social Network Analysis with Python*,
Computer Communications and Networks,
https://doi.org/10.1007/978-3-319-96746-2_13

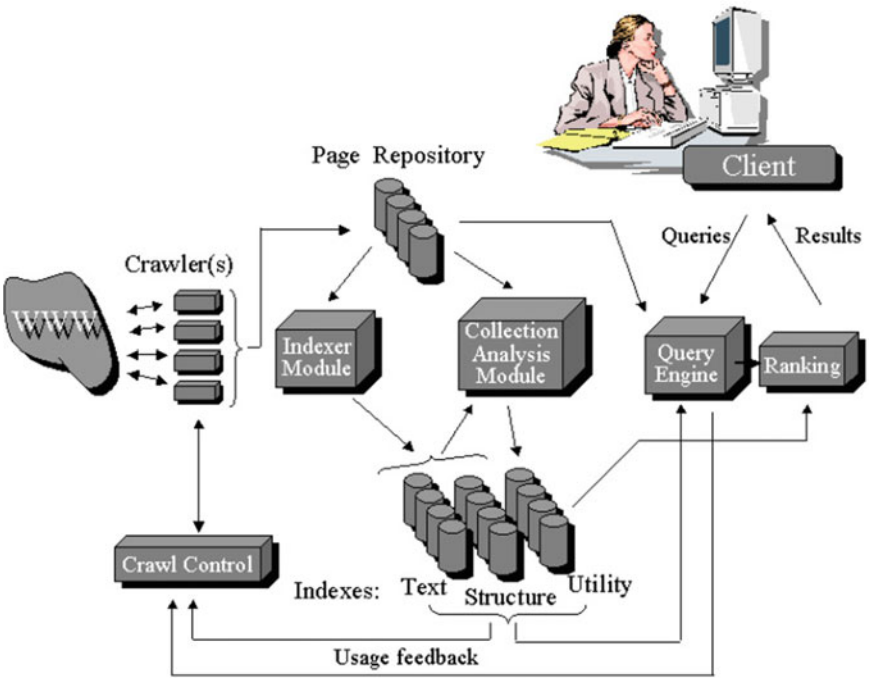

Fig. 13.1 Typical search engine architecture

13.1.1 Crawling

The crawler module starts with an initial set of URLs S_0 which are initially kept in a priority queue. From the queue, the crawler gets a URL, downloads the page, extracts any URLs in the downloaded page, and puts the new URLs in the queue. This process repeats until the *crawl control* module asks the crawler to stop.

13.1.1.1 Page Selection

Due to the sheer quantity of pages in the Web and limit in the available resources, the crawl control program has to decide which pages to crawl, and which not to. The "importance" of a page can be defined in one of the following ways:

1. *Interest Driven*: The important pages can be defined as those of "interest to" the users. This could be defined in the following manner: given a query Q, the importance of the page P is defined as the textual similarity between P and Q [1]. This metric is referred to as $IS(\cdot)$.
2. *Popularity Driven*: Page importance depends on how "popular" a page is. Popularity can be defined in terms of a page's in-degree. However, a page's popularity

is the in-degree with respect to the entire Web. This can be denoted as $IB(\cdot)$. The crawler may have to instead provide an estimate $IB'(\cdot)$ with the in-degree from the pages seen so far.
3. *Location Driven*: This importance measure is computed as a function of a page's location instead of its contents. Denoted as $IL(\cdot)$, if URL u leads to a page P, then $IL(P)$ is a function of u.

Therefore, the importance of a page P can be defined as $IC(P) = k_1 IS(P) + k_2 IB(P) + k_3 IL(P)$, for constants k_1, k_2, k_3 and query Q.

Next, we will look at the performance of a crawler. The following ways may be used to compute the performance:

1. *Crawl and stop*: Under this method, a crawler C starts at a page P_0 and stops after visiting K pages. Here, K denotes the number of pages that the crawler can download in one crawl. At this point a perfect crawler would have visited $R_1, \ldots, R_K$, where R_1 is the page with the highest importance value, R_2 is the next highest, and so on. These pages R_1 through R_K are called the *hot pages*. The K pages visited by our real crawler will contain only $M(\leq K)$ pages with rank higher than or equal to that of R_K. We need to know the exact rank of all pages in order to obtain the value M. The performance of the crawler C is computed as $P_{CS}(C) = (M \cdot 100)/K$. Although the performance of the perfect crawler is 100%, a crawler that manages to visit pages at random would have a performance of $(K \cdot 100)/T$, where T is the total number of pages in the Web.
2. *Crawl and stop with threshold*: Assume that a crawler visits K pages. If we are given an importance target G, then any page with importance target greater than G is considered hot. If we take the total number of hot pages to be H, then the performance of the crawler, $P_{ST}(C)$, is the percentage of the H hot pages that have been visited when the crawler stops. If $K < H$, then an ideal crawler will have performance $(K \cdot 100)/H$. If $K \geq H$, then the ideal crawler has 100% performance. A purely random crawler that revisits pages is expected to visit $(H/T)K$ hot pages when it stops. Thus, its performance is $(k \cdot 100)/T$. Only if the random crawler visits all T pages, is its performance expected to be 100%.

13.1.1.2 Page Refresh

Once the crawler has downloaded the "important" pages, these pages must be periodically refreshed to remain up-to-date. The following strategies can be used to refresh the pages:

1. *Uniform refresh policy*: All pages are revisited at the same frequency f, regardless of how often they change.
2. *Proportional refresh policy*: The crawler visits a page proportional to its change. If λ_i is the change frequency of a page e_i, and f_i is the crawler's revisit frequency for e_i, then the frequency ratio λ_i/f_i is the same for any i. Keep in mind that the crawler needs to estimate λ_i's for each page, in order to implement this policy.

This estimation can be based on the change history of a page that the crawler can collect [9].

Reference [3] defines the "freshness" and "age" of a Webpage as follows:

1. *Freshness*: Let $S = \{e_1, \ldots, e_N\}$ be the collection of N pages. The *freshness* of a local page e_i at time t is as given in Eq. 13.1

$$F(e_i; t) = \begin{cases} 1 \; if \; e_i \; is \; up - to - date \; at \; time \; t \\ 0 \; otherwise \end{cases} \tag{13.1}$$

Then the *freshness* of the local collection S at time t is as given in Eq. 13.2.

$$F(S; t) = \frac{1}{N} \sum_{i=1}^{N} F(e_i; t) \tag{13.2}$$

Since the pages get refreshed over time, the time average of the freshness of page e_i, $\overline{F}(e_i)$, and the time average of the freshness of collection S, $\overline{F}(S)$, are defined as in Eqs. 13.3 and 13.4.

$$\overline{F}(e_i) = \lim_{t \to \infty} \frac{1}{t} \int_0^t F(e_i; t) dt \tag{13.3}$$

$$\overline{F}(S) = \lim_{t \to \infty} \frac{1}{t} \int_0^t F(S; t) dt \tag{13.4}$$

2. *Age*: The *age* of the local page e_i at time t is as given in Eq. 13.5.

$$A(e_i; t) = \begin{cases} 0 \; if \; e_i \; is \; up - to - date \; at \; time \; t \\ t - modification \; time \; of \; e_i \; otherwise \end{cases} \tag{13.5}$$

Then the *age* of the local collection S at time t is as given in Eq. 13.6.

$$A(S; t) = \frac{1}{N} \sum_{i=1}^{N} A(e_i; t) \tag{13.6}$$

13.1.2 Storage

The storage repository of a search engine must perform two basic functions. First, it must provide an interface for the crawler to store pages. Second, it must provide an efficient access API that the indexer module and the collection analysis module can use to retrieve pages.

The issues a repository must deal with are as follows: First, it must be *scalable*, i.e, it must be capable of being distributed across a cluster of systems. Second, it must be capable of supporting both *random access* and *streaming access* equally efficiently. Random access for quickly retrieving a specific Webpage, given the page's unique identifier to serve out cached copies to the end-user. Streaming access to receive the entire collection as a stream of pages to provide pages in bulk for the indexer and analysis module to process and analyse. Third, since the Web changes rapidly, the repository needs to handle a high rate of modifications since pages have to be refreshed periodically. Lastly, the repository must have a mechanism of detecting and removing obsolete pages that have been removed from the Web.

A distributed Web repository that is designed to function over a cluster of interconnected *storage nodes* must deal with the following issues that affect the characteristics and performance of the repository. A detailed explanation of these issues are found in [15].

13.1.2.1 Page Distribution Policies

Pages can be assigned to nodes using a number of different policies. In a *uniform distribution policy*, all nodes are treated identically, and a page can be assigned to any of these nodes, independent of its identifier. Thus each node will store portions of the collection proportionate to its storage capacity. On the other hand, a *hash distribution policy* allocates pages to nodes depending on the page identifiers. A page identifier would be hashed to yield a node identifier and the page would be allocated to the corresponding node.

13.1.2.2 Physical Page Organization Methods

In essence, physical page organization at each node determines how well each node supports *page addition/insertion*, *high-speed streaming* and *random page access*.

A hash-based organization treats a disk as a set of hash-buckets, each of which is small enough to fit in memory. Pages are assigned to hash buckets depending on their page identifier. For page additions, a log-structured organization may be used in which the entire disk is treated as a large contiguous log to which incoming pages are appended. Random access is supported using a separate B-tree index that maps page identifiers to physical locations on disk. One can also devise a hybrid hashed-log organization, where storage is divided into large sequential "extents", as opposed to buckets that fit in memory. Pages are hashed into extents, and each extent is organized like a log-structured file.

13.1.2.3 Update Strategies

The updates of the crawler can be structured in two ways:

1. *Batch-mode or steady crawler*: A batch-mode crawler is periodically executed and allowed to crawl for a certain amount of time or until a targeted set of pages have been crawled, and then stopped. In contrast, a steady crawler runs continuously, without pause, supplying updates and new pages to the repository.
2. *Partial or complete crawls*: A partial crawl recrawls only a specific set of pages, while a complete crawl performs a total crawl of all the pages. This makes sense only for a batch- mode crawler while a steady crawler cannot make such a distinction.
3. *In-place update or shadowing*: With in-place updates, pages received from the crawl are directly integrated into the repository's existing collection, probably replacing the older version. With shadowing, pages from a crawl are stored separate from the existing collection and updates are applied in a separate step.

   The advantage of shadowing is that the reading and updating tasks can be delegated to two separate nodes thereby ensuring that a node does not have to concurrently handle both page addition and retrieval. However, this is a two-edged sword. While avoiding conflict by simplifying implementation and improving performance, there is a delay between the time a page is retrieved by the crawler and the time it is available for access.

13.1.3 Indexing

The indexer and the collection analysis modules are responsible for generating the text, structure and the utility indexes. Here, we describe the indexes.

13.1.3.1 Structure Index

To build this index, the crawled content is modelled as a graph with nodes and edges. This index helps search algorithms by providing *neighbourhood information*: For a page P, retrieve the set of pages pointed to by P or the set of pages pointing to P, and *sibling pages*: Pages related to a given page P.

An *inverted index* over a collection of Webpages consists of a set of *inverted lists*, one for each word. The inverted list for a term is a sorted collection of locations (page identifier and the position of the term in the page) where the term appears in the collection.

To make the inverted index scalable, there are two basic strategies for partitioning the inverted index across a collection of nodes.

In the *local inverted file* (IF_L) organization [21], each node is responsible for a disjoint subset of pages in the collection. A search query would be broadcast to all nodes, each of which would return disjoint lists of page identifiers containing the search terms.

Global inverted file (IF_G) organization [21] partitions on index terms so that each query server stores inverted lists only for a subset of the terms in the collection.

Reference [19] describes important characteristics of the IF_L strategy that make this organization ideal for the Web search environment. Performance studies in [22] indicate that IF_L organization uses system resources effectively and provides good query throughput in most cases.

13.1.3.2 Text Index

Traditional information retrieval methods using suffix arrays, inverted indices and signature files, can be be used to generate these text index.

13.1.3.3 Utility Index

The number and type of utility indexes built by the collection analysis module depends on the features of the query engine and the type of information used by the ranking module.

13.1.4 Ranking

The query engine collects search terms from the user and retrieves pages that are likely to be relevant. These pages cannot be returned to the user in this format and must be ranked.

However, the problem of ranking is faced with the following issues. First, before the task of ranking pages, one must first retrieve the pages that are relevant to a search. This information retrieval suffers from problems of *synonymy* (multiple terms that more or less mean the same thing) and *polysemy* (multiple meanings of the same term, so by the term "jaguar", do you mean the animal, the automobile company, the American football team, or the operating system).

Second, the time of retrieval plays a pivotal role. For instance, in the case of an event such as a calamity, government and news sites will update their pages as and when they receive information. The search engine not only has to retrieve the pages repeatedly, but will have to re-rank the pages depending on which page currently has the latest report.

Third, with every one capable of writing a Web page, the Web has an abundance of information for any topic. For example, the term "social network analysis" returns a search of around 56 million results. Now, the task is to rank these results in a manner such that the most important ones appear first.

13.1.4.1 HITS Algorithm

When you search for the term "msrit", what makes www.msrit.edu a good answer? The idea is that the page www.msrit.edu is not going to use the term "msrit" more frequently or prominently than other pages. Therefore, there is nothing in the page that makes it stand out in particular. Rather, it stands out because of features on other Web pages: when a page is relevant to "msrit", very often www.msrit.edu is among the pages it links to.

One approach is to first retrieve a large collection of Web pages that are relevant to the query "msrit" using traditional information retrieval. Then, these pages are voted based on which page on the Web receives the greatest number of in-links from pages that are relevant to msrit. In this manner, there will ultimately be a page that will be ranked first.

Consider the search for the term "newspapers". Unlike the case of "msrit", the position for a good answer for the term "newspapers" is not necessarily a single answer. If we try the traditional information retrieval approach, then we will get a set of pages variably pointing to several of the news sites.

Now we attempt to tackle the problem from another direction. Since finding the best answer for a query is incumbent upon the retrieved pages, finding good retrieved pages will automatically result in finding good answers. Figure 13.2 shows a set of retrieved pages pointing to newspapers.

We observe from Fig. 13.2 that among the sites casting votes, a few of them voted for many of the pages that received a lot of votes. Therefore, we could say that these pages have a sense where the good answers are, and to score them highly as lists. Thus, a page's value as a list is equal to the sum of the votes received by all pages that it voted for. Figure 13.3 depicts the result of applying this rule to the pages casting votes.

If pages scoring well as lists are believed to actually have a better sense for where the good results are, then we should weigh their votes more heavily. So, in particular, we could tabulate the votes again, but this time giving each page's vote a weight equal to its value as a list. Figure 13.4 illustrates what happens when weights are accounted for in the newspaper case.

This re-weighting can be done repeatedly.

The "good" answers that we were originally seeking for a query are called the *authorities* for the query and the high-value lists are called the *hubs* for the query. The goal here is to estimate a page p's value as a potential authority and as a potential hub. Therefore, each page p is assigned two numerical scores: $auth(p)$ and $hub(p)$. Each of these initially starts out at 1.

Now, $auth(p)$ and $hub(p)$ are updated according to the *Authority Update rule* and *Hub Update rule* respectively. The authority update rule states that for each page p, $auth(p)$ is computed as the sum of the hub scores of all pages that point to it. The hub update rule says that for each page p, $hub(p)$ is the sum of the authority scores of all pages that it points to.

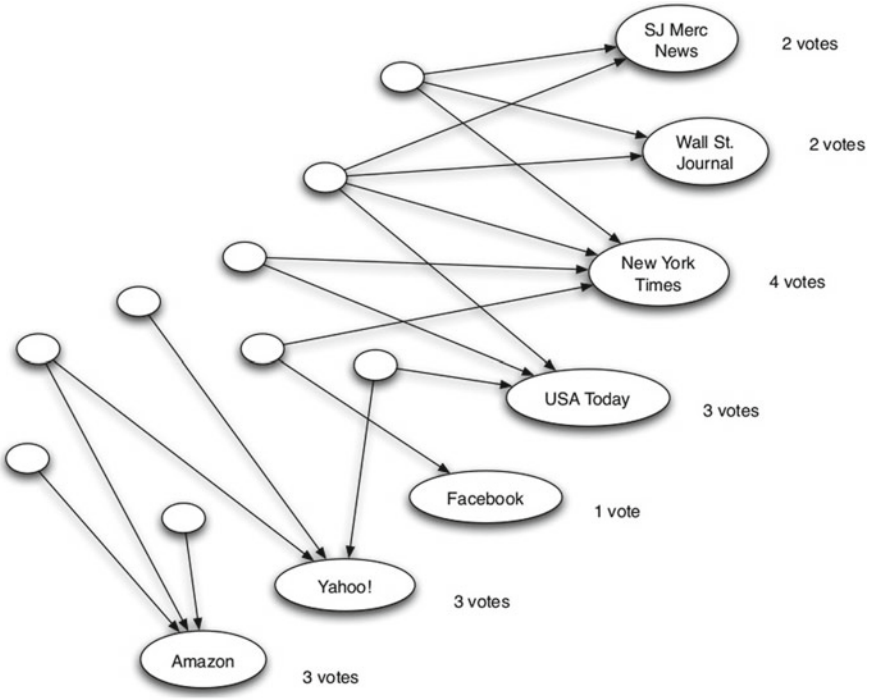

Fig. 13.2 Counting in-links to pages for the query "newspapers"

By the principle of repeated improvement, we do the following:

1. Start with all hub and authority scores equal to 1.
2. Choose a number of steps k.
3. Perform a sequence of k hub-authority updates. Each update works as follows:
   - Apply the Authority Update rule to the current set of scores.
   - Apply the Hub Update rule to the resulting set of scores.
4. The resultant hub and authority scores may involve numbers that are very large (as can be seen in Fig. 13.4). But we are concerned only with their relative size and therefore they can be normalized: we divide down each authority score by the sum of all authority scores, and divide down each hub score by the sum of all hub scores. Figure 13.5 shows the normalized scores of Fig. 13.4.

This algorithm is commonly known as the HITS algorithm [16].

However, as k goes to infinity, the normalized values converge to a limit. Figure 13.6 depicts the limiting values for the "newspaper" query.

These limiting values correspond to a state of equilibrium where a page p's authority score is proportional to the hub scores of the pages that point to p, and p's hub score is proportional to the authority scores of the pages p points to.

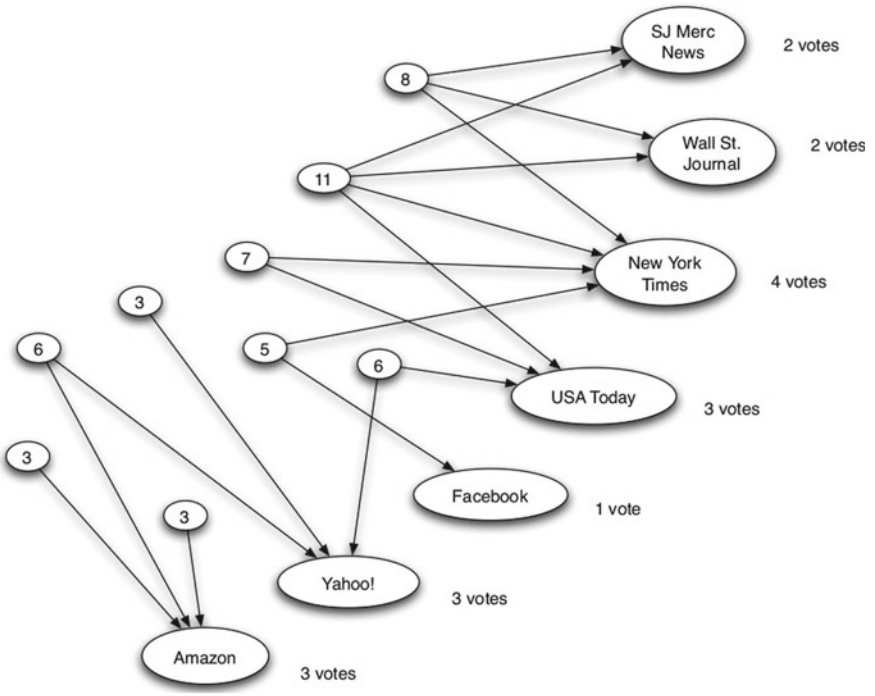

Fig. 13.3 Finding good lists for the query "newspapers": each page's value as a list is written as a number inside it

When we intend to perform the spectral analysis of HITS algorithm, we will first represent the network of n nodes as an adjacency matrix M. Although, it is not a very efficient way to represent a large network (as studied in Chap. 1), it is conceptually useful for analysis.

The hub and authority scores are represented as vectors in n dimensions. The vector of hub scores is denoted as h where the ith element represents the hub score of node i. On similar lines, we denote the vector of authority scores as a. Thus, the Hub Update rule can be represented by Eq. 13.7 and therefore the Authority Update rule can be represented by Eq. 13.8.

$$h \leftarrow M \cdot a \tag{13.7}$$

$$a \leftarrow M^T \cdot h \tag{13.8}$$

The k-step hub-authority computations for large values of k is be described below:

The initial authority and hub vectors are denoted as $a^{[0]}$ and $h^{[0]}$ each of which are equal to unit vectors. Let $a^{[k]}$ and $h^{[k]}$ represent the vectors of authority and hub

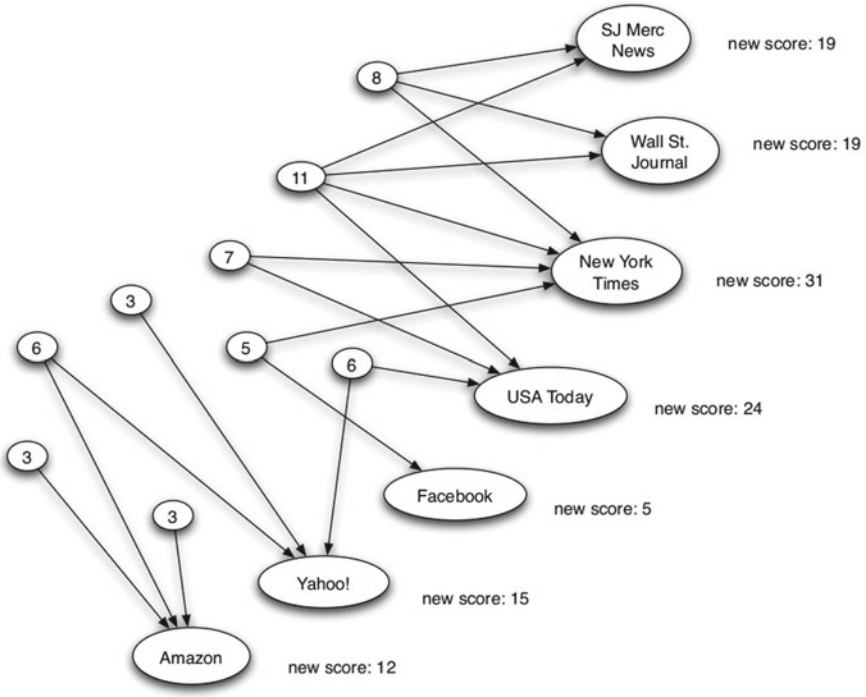

Fig. 13.4 Re-weighting votes for the query "newspapers": each of the labelled page's new score is equal to the sum of the values of all lists that point to it

scores after k applications of the Authority and Hub Update Rules in order. This gives us Eqs. 13.9 and 13.10.

$$a^{[1]} = M^T h^{[0]} \tag{13.9}$$

and

$$h^{[1]} = Ma^{[1]} = MM^T h^{[0]} \tag{13.10}$$

In the next step, we get

$$a^{[2]} = M^T h^{[1]} = M^T MM^T h^{[0]} \tag{13.11}$$

and

$$h^{[2]} = Ma^{[2]} = MM^T MM^T h^{[0]} = (MM^T)^2 h^{[0]} \tag{13.12}$$

For larger values of k, we have

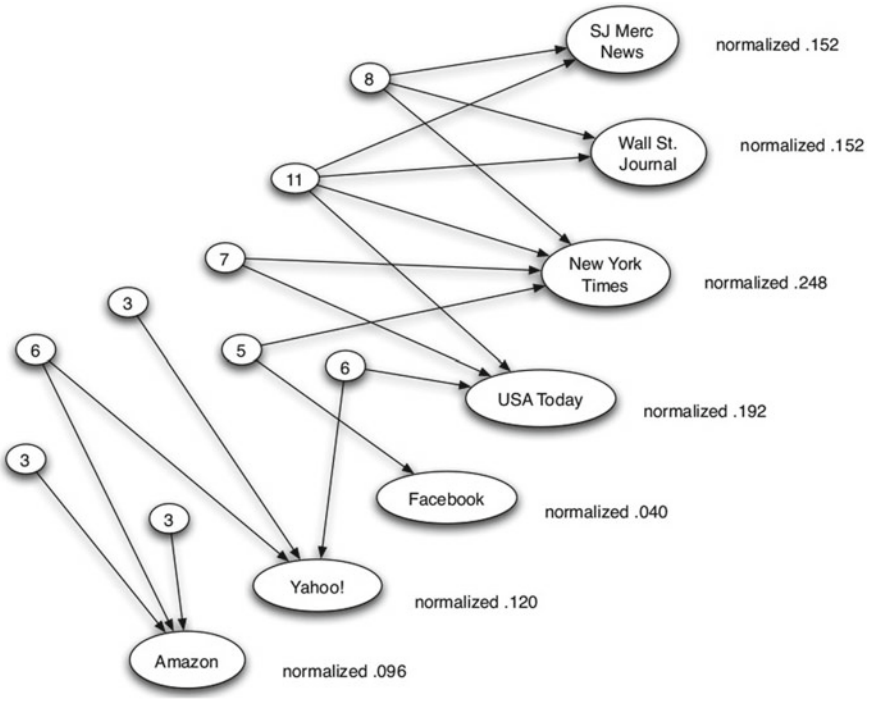

Fig. 13.5 Re-weighting votes after normalizing for the query "newspapers"

$$a^{[k]} = (M^T M)^{k-1} M^T h^{[0]} \tag{13.13}$$

and

$$h^{[k]} = (MM^T)^k h^{[0]} \tag{13.14}$$

Now we will look at the convergence of this process.

Consider the constants c and d. If

$$\frac{h^{[k]}}{c^k} = \frac{(MM^T)^k h^{[0]}}{c^k} \tag{13.15}$$

is going to converge to a limit $h^{[*]}$, we expect that at the limit, the direction of $h^{[*]}$ shouldn't change when it is multiplied by (MM^T), although its length might grow by a factor of c, i.e, we expect that h^* will satisfy the equation

$$(MM^T)h^{[*]} = ch^{[*]} \tag{13.16}$$

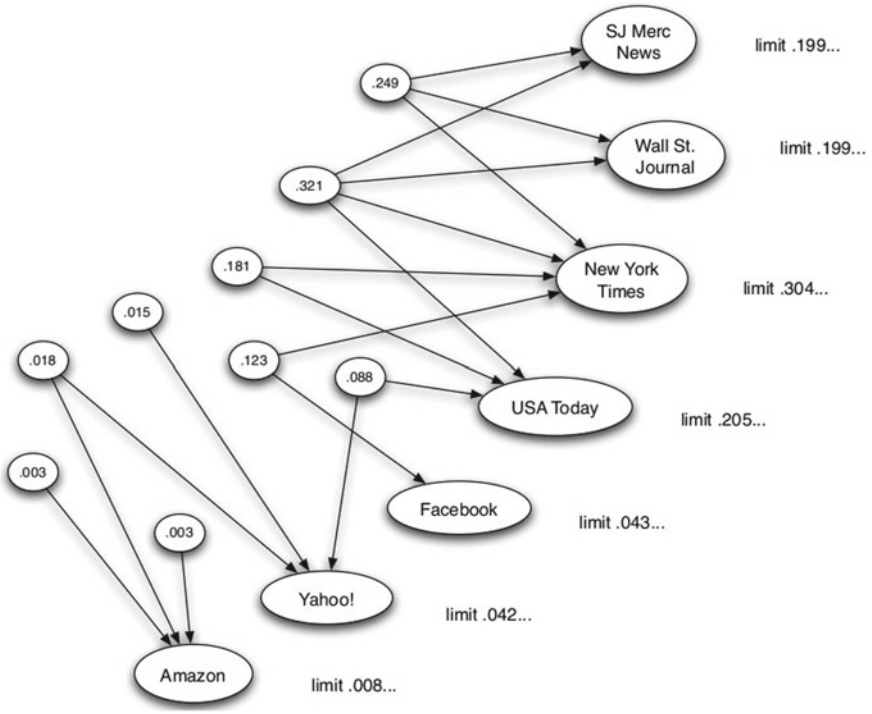

Fig. 13.6 Limiting hub and authority values for the query "newspapers"

This means that $h^{[*]}$ is the eigenvector of the matrix MM^T, with c a corresponding eigenvalue.

Any symmetric matrix with n rows and n columns has a set of n eigenvectors that are all unit vectors and all mutually orthogonal, i.e, they form a basis for the space $\mathbb{R}^n$.

Since MM^T is symmetric, we get the n mutually orthogonal eigenvectors as $z_1, \ldots, z_n$, with corresponding eigenvalues $c_1, \ldots, c_n$. Let $|c_1| \geq |c_2| \geq \cdots \geq |c_n|$. Suppose that $|c_1| > |c_2|$, if we have to compute the matrix-vector product $(MM^T)x$ for any vector x, we could first write x as a linear combination of the vectors $z_1, \ldots, z_n$, i.e, $x = p_1 z_1 + p_2 z_2 + \cdots + p_n z_n$ for coefficients $p_1, \ldots, p_n$. Thus

$$\begin{aligned}(MM^T)x &= (MM^T)(p_1 z_1 + p_2 z_2 + \cdots + p_n z_n) \\ &= p_1 MM^T z_1 + p_2 MM^T z_2 + \cdots + p_n MM^T z_n \\ &= p_1 c_1 z_1 + p_2 c_2 z_2 + \cdots + p_n c_n z_n\end{aligned} \tag{13.17}$$

k multiplications of MM^T of the matrix-vector product will therefore give us

$$(MM^T)^k x = c_1^k p_1 z_1 + c_2^k p_2 z_2 + \cdots + c_n^k p_n z_n \tag{13.18}$$

Applying Eq. 13.18 to the vector of hub scores, we get $h^{[k]} = (MM^T)h^{[0]}$. Since $h^{[0]}$ is just a starting unit vector, it can be represented in terms of the basis vectors $z_1, \ldots, z_n$ as some linear combination $h^{[0]} = q_1 z_1 + q_2 z_2 + \cdots + q_n z_n$. So,

$$h^{[k]} = (MM^T)^k h^{[0]} = c_1^k q_1 z_1 + c_2^k q_2 z_2 + \cdots + c_n^k q_n z_n \tag{13.19}$$

Dividing both sides by c_1^k, we get

$$\frac{h^{[k]}}{c_1^k} = q_1 z_1 + \left(\frac{c_2}{c_1}\right)^k q_2 z_2 + \cdots + \left(\frac{c_n}{c_1}\right)^k q_n z_n \tag{13.20}$$

Since $|c_1| > |c_2|$, as k goes to infinity, every term on the right-hand side except the first is going to 0. This means that the sequence of vectors $\frac{h^{[k]}}{c_1^k}$ converges to the limit $q_1 z_1$ as k goes to infinity.

What would happen if we began the computation from some starting vector x, instead of the unit vector $h^{[0]}$? If x is a positive vector expressed as $x = p_1 z_1 + \cdots + p_n z_n$ for some coefficients $p_1, \ldots, p_n$, and so $(MM^T)^k x = c_1^k p_1 z_1 + \cdots + c_n^k p_n z_n$. Then, $\frac{h^{[k]}}{c_1^k}$ converges to $p_1 z_1$. Thus, we converge to a vector in the direction of z_1 even with this new starting vector.

Now we consider the case when the assumption $|c_1| > |c_2|$ is relaxed. Let's say that there are $l > 1$ eigenvalues that are tied for the largest absolute value, i.e, $|c_1| = \cdots = |c_l|$, and then $c_{l+1}, \ldots, c_n$, are all smaller in absolute value. With all the eigenvalues in MM^T being non-negative, we have $c_1 = \cdots = c_l > c_{l+1} \geq \cdots \geq c_n \geq 0$. This gives us

$$\frac{h^{[k]}}{c_1^k} = \frac{c_1^k q_1 z_1 + \cdots + c_n^k q_n z_n}{c_1^k} = q_1 z_1 + \cdots + q_l z_l + \left(\frac{c_{l+1}}{c_l}\right)^k q_{l+1} z_{l+1} + \cdots + \left(\frac{c_n}{c_1}\right)^k q_n z_n \tag{13.21}$$

Terms $l + 1$ through n of this sum go to zero, and so the sequence converges to $q_1 z_1 + \cdots + q_l z_l$. Thus, when $c_1 = c_2$, we still have convergence, but the limit to which the sequence converges might now depend on the choice of the initial vector $h^{[0]}$ (and particularly its inner product with each of $z_1, \ldots, z_l$). In practice, with real and sufficiently large hyperlink structures, one essentially always gets a matrix M with the property that MM^T has $|c_1| > |c_2|$.

This can be adapted directly to analyse the sequence of authority vectors. For the authority vectors, we are looking at powers of $(M^T M)$, and so the basic result is that the vector of authority scores will converge to an eigenvector of the matrix $M^T M$ associated with its largest eigenvalue.

Table 13.1 PageRank values of Fig. 13.7

Step	A	B	C	D	E	F	G	H
1	1/2	1/16	1/16	1/16	1/16	1/16	1/16	1/8
2	3/16	1/4	1/4	1/32	1/32	1/32	1/32	1/16

13.1.4.2 PageRank

PageRank [7] also works on the same principle as HITS algorithm. It starts with simple voting based on in-links, and refines it using the Principle of Repeated Improvement.

PageRank can be thought of as a "fluid" that circulates through the network, passing from node to node across edges, and pooling at the nodes that are most important. It is computed as follows:

1. In a network with n nodes, all nodes are assigned the same initial PageRank, set to be $1/n$.
2. Choose a number of steps k.
3. Perform a sequence of k updates to the PageRank values, using the following *Basic PageRank Update rule* for each update: Each page divides its current PageRank equally across its out-going links, and passes these equal shares to the pages it points to. If a page has no out-going links, it passes all its current PageRank to itself. Each page updates its new PageRank to be the sum of the shares it receives.

This algorithm requires no normalization because the PageRank values are just moved around with no chance of increase.

Figure 13.7 is an instance of a network with eight Webpages. All of these pages start out with a PageRank of $1/8$. Their PageRanks after the first two updates are as tabulated in Table 13.1.

Similar to the HITS algorithm, the PageRank values converge to a limit as k goes to infinity. Figure 13.8 depicts the equilibrium PageRank values.

There is a problem with the PageRank rule in this form. Consider the network in Fig. 13.9, which is the same network as in Fig. 13.7 but with F and G pointing to one another rather than pointing to A. This causes the PageRanks to converge to $1/2$ for F and G with 0 for all other nodes.

This means that any network containing reciprocating links will have this clogging of PageRanks at such nodes. To prevent such a situation, the PageRank rule is updated by introducing a scaling factor s that should be strictly between 0 and 1. This gives us the following *Scaled PageRank Update rule*. According to this rule, first apply the Basic PageRank Update rule. Then scale down all PageRank values by a factor of s, thereby shrinking the total PageRank from 1 to s. The residual $1 - s$ units of PageRank are divided equally over all the nodes, giving $(1 - s)/n$ to each.

This scaling factor makes the PageRank measure less sensitive to the addition or deletion of small number of nodes or edges.

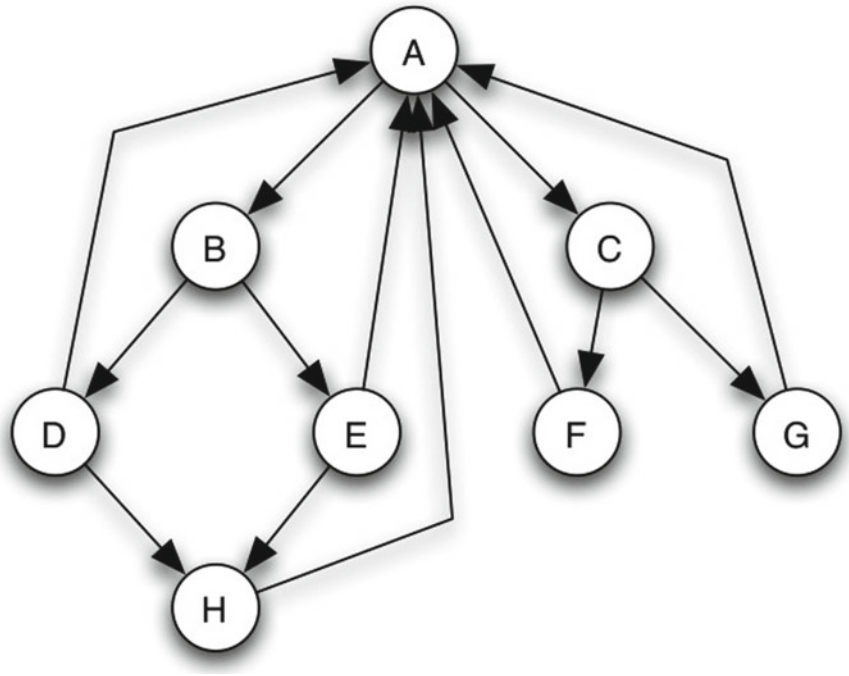

Fig. 13.7 A collection of eight web pages

A survey of the PageRank algorithm and its developments can be found in [4].

We will first analyse the Basic PageRank Update rule and then move on to the scaled version. Under the basic rule, each node takes its current PageRank and divides it equally over all the nodes it points to. This suggests that the "flow" of PageRank specified by the update rule can be naturally represented using a matrix N. Let N_{ij} be the share of i's PageRank that j should get in one update step. $N_{ij} = 0$ if i doesn't link to j, and when i links to j, then $N_{ij} = 1/l_i$, where l_i is the number of links out of i. (If i has no outgoing links, then we define $N_{ii} = 1$, in keeping with the rule that a node with no outgoing links passes all its PageRank to itself.)

If we represent the PageRank of all the nodes using a vector r, where r_i is the PageRank of node i. In this manner, the Basic PageRank Update rule is

$$r \leftarrow N^T \cdot r \tag{13.22}$$

We can similarly represent the Scaled PageRank Update rule using the matrix $\hat{N}$ to denote the different flow of PageRank. To account for the scaling, we define $\hat{N}_{ij}$ to be $sN_{ij} + (1-s)/n$, this gives the scaled update rule as

$$r \leftarrow \hat{N}^T \cdot r \tag{13.23}$$

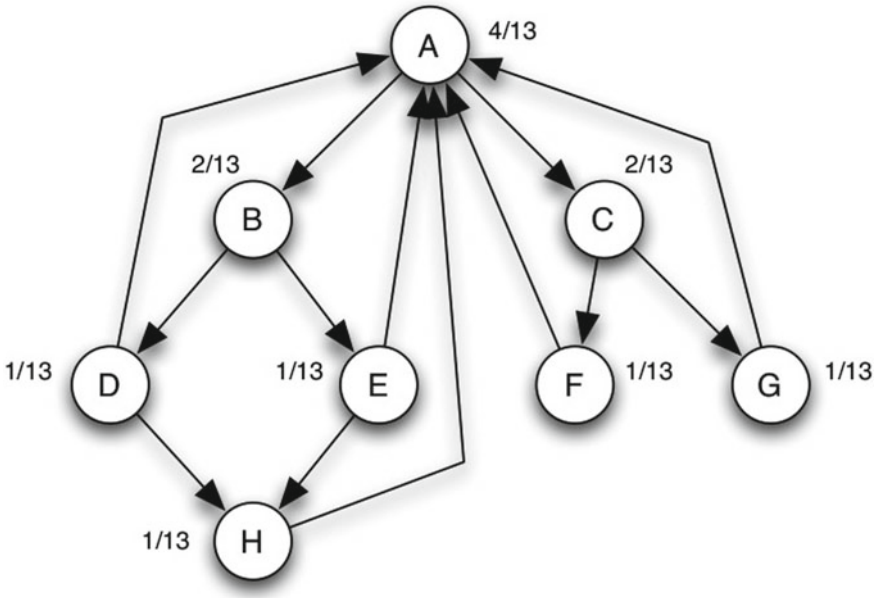

Fig. 13.8 Equilibrium PageRank values for the network in Fig. 13.7

Starting from an initial PageRank vector $r^{[0]}$, a sequence of vectors $r^{[1]}, r^{[2]}, \ldots$ are obtained from repeated improvement by multiplying the previous vector by $\hat{N}^T$. This gives us

$$r^{[k]} = (\hat{N}^T)^k r^{[0]} \tag{13.24}$$

This means that if the Scaled PageRank Update rule converges to a limiting vector $r^{[*]}$, this limit would satisfy $\hat{N}^T r^{[*]} = r^{[*]}$. This is proved using *Perron's Theorem* [18].

Reference [2] describes axioms that are satisfied by the PageRank algorithm, and that any page ranking algorithm that satisfies these must coincide with the PageRank algorithm.

13.1.4.3 Random Walk

Consider the situation where one randomly starts browsing a network of Webpages. They start at a random page and pick each successive page with equal probability. The links are followed for a sequence of k steps: in each step, a random out-going link from the current page is picked. (If the current page has no out-going links, they just stay where they are.)

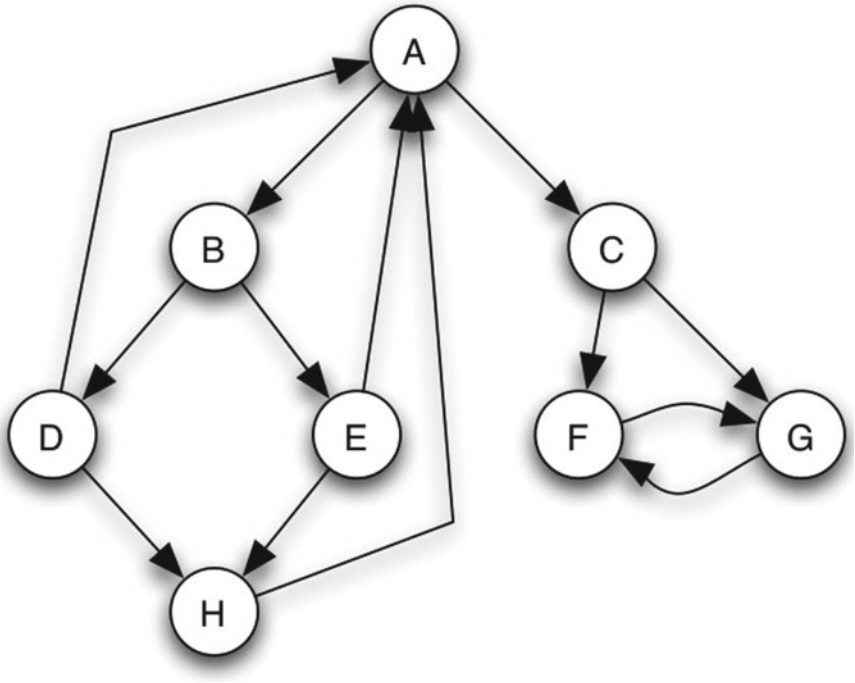

Fig. 13.9 The same collection of eight pages, but F and G have changed their links to point to each other instead of to A. Without a smoothing effect, all the PageRank would go to F and G

Remark 1 Taking a random walk in this situation, the probability of being at a page X after k steps of this random walk is precisely the PageRank of X after k applications of the Basic PageRank Update rule.

Proof If $b_1, b_2, \ldots, b_n$ denote the probabilities of the walk being at nodes $1, 2, \ldots, n$ respectively in a given step, then the probability it will be at node i in the next step is computed as follows:

1. For each node j that links to i, if we are given that the walk is currently at node j, then there is a $1/l_j$ chance that it moves from j to i in the next step, where l_j is the number of links out of j.
2. The walk has to actually be at node j for this to happen, so node j contributes $b_j(1/l_j) = b_j/l_j$ to the probability of being at i in the next step.
3. Therefore, summing b_j/l_j over all nodes j that link to i gives the probability the walk is at b_i in the next step.

So the overall probability that the walk is at i in the next step is the sum of b_j/l_j over all nodes that link to i.

If we represent the probabilities of being at different nodes using a vector b, where the coordinate b_i is the probability of being at node i, then this update rule can be written using matrix-vector multiplication as

$$b \leftarrow N^T \cdot b \tag{13.25}$$

This is exactly the same as Eq. 13.22. Since both PageRank values and random-walk probabilities start out the same (they are initially $1/n$ for all nodes), and they then evolve according to exactly the same rule, so they remain same forever. This justifies the claim.

Remark 2 The probability of being at a page X after k steps of the scaled random walk is precisely the PageRank of X after k applications of the Scaled PageRank Update Rule.

Proof We go by the same lines as the proof of Remark 1. If $b_1, b_2, \ldots, b_n$ denote the probabilities of the walk being at nodes $1, 2, \ldots, n$ respectively in a given step, then the probability it will be at node i in the next step, is the sum of sb_j/l_j, over all nodes j that link to i, plus $(1-s)/n$. If we use the matrix $\hat{N}$, then we can write the probability update as

$$b \leftarrow \hat{N^T} b \tag{13.26}$$

This is the same as the update rule from Eq. 13.23 for the scaled PageRank values. The random-walk probabilities and the scaled PageRank values start at the same initial values, and then evolve according to the same update, so they remain the same forever. This justifies the argument.

A problem with both PageRank and HITS is *topic drift*. Because they give the same weights to all edges, the pages with the most in-links in the network being considered tend to dominate, whether or not they are most relevant to the query. References [8] and [5] propose heuristic methods for differently weighting links. Reference [20] biased PageRank towards pages containing a specific word, and [14] proposed applying an optimized version of PageRank to the subset of pages containing the query terms.

13.1.4.4 SALSA Algorithm

Reference [17] proposed the SALSA algorithm which starts with a "Root Set" short list of Webpages relevant to the given query retrieved from a text-based search engine. This root set is augmented by pages which link to pages in the Root Set, and also pages which are linked to pages in the Root Set, to obtain a larger "Base Set" of Webpages. Perform a random walk on this base set by alternately (a) going uniformly to one of the pages which links to the current page, and (b) going uniformly to one of the pages linked to by the current page. The authority weights are defined to be the stationary

distribution of the two-step chain first doing (a) and then (b), while the hub weights are defined to be the stationary distribution of the two-step chain first doing (b) and then (a).

Formally, let $B(i) = \{k : k \rightarrow i\}$ denote the set of all nodes that point to i, i.e, the nodes we can reach from i by following a link backwards, and let $F(i) = \{k : i \rightarrow k\}$ denote the set of all nodes that we can reach from i by following a forward link. The Markov Chain for the authorities has transition probabilities

$$P_a(i, j) = \sum_{k:k \in B(i) \cap B(j)} \frac{1}{|B(i)|} \frac{1}{|F(k)|} \tag{13.27}$$

If the Markov chain is irreducible, then the stationary distribution $a = (a_1, a_2, \ldots, a_N)$ of the Markov chain satisfies $a_i = |B(i)|/|B|$, where $B = \bigcup_i B(i)$ is the set of all backward links.

Similarly, the Markov chain for the hubs has transition probabilities

$$P_h(i, j) = \sum_{k:k \in F(i) \cap F(j)} \frac{1}{|F(i)|} \frac{1}{|B(k)|} \tag{13.28}$$

and the stationary distribution $h = (h_1, h_2, \ldots, h_N)$ of the Markov chain satisfies $h_i = |F(i)|/|F|$, where $F = \bigcup_i F(i)$ is the set of all forward links.

In the special case of a single component, SALSA can be viewed as a one-step truncated version of HITS algorithm, i.e, in the first iteration of HITS algorithm, if we perform the Authority Update rule first, the authority weights are set to $a = A^T u$, where u is the vector of all ones. If we normalize in the L_1 norm, then $a_i = |B(i)|/|B|$, which is the stationary distribution of the SALSA algorithm. A similar observation can be made for the hub weights.

If the underlying graph of the Base Set consists of more than one component, then the SALSA algorithm selects a starting point uniformly at random, and performs a random walk within the connected component that contains that node. Formally, let j be a component that contains node i, let N_j denote the number of nodes in the component, and B_j the set of (backward) links in component j. Then, the authority weight of node i is

$$a_i = \frac{N_j}{N} \frac{|B(i)|}{|B_j|} \tag{13.29}$$

A simplified version of the SALSA algorithm where the authority weight of a node is the ratio $|B(i)|/|B|$, corresponds to the case that the starting point for the random walk is chosen with probability proportional to its in-degree. This variation of the SALSA algorithm is commonly referred to as pSALSA.

13.1.4.5 PHITS Algorithm

Reference [10] proposed a statistical hubs and authorities algorithm called PHITS algorithm. A probabilistic model was proposed in which a citation c of a document d is caused by a latent "factor" or "topic", z. It is postulated that there are conditional distributions $P(c|z)$ of a citation c given a factor z, and also conditional distributions $P(z|d)$ of a factor z given a document d. In terms of these conditional distributions, they produce a likelihood function.

The EM Algorithm of [11] to assign the unknown conditional probabilities so as to maximize this likelihood function L, best explained the proposed data. Their algorithm requires specifying in advance the number of factors z to be considered. Furthermore, it is possible that the EM algorithm could get stuck in a local maximum, without converging to the true global maximum.

13.1.4.6 Breadth First Algorithm: A Normalized n-step Variant

Reference [6] proposed a BFS algorithm, as a generalization of the pSALSA algorithm, and a restriction of the HITS algorithm. The BFS algorithm extends the idea of in-degree that appears in pSALSA from a one link neighbourhood to a n-link neighbourhood. Here, we introduce the following notations. If we follow a link backwards, then we call this a B path, and if we follow a link forwards, we call it a F path. So, a BF path is a path that first follows a link backwards, and then a link forward. $(BF)^n(i, j)$ denotes the set of $(BF)^n$ paths that go from i to j, $(BF)^n(i)$ the set of $(BF)^n$ paths that leave node i, and $(BF)^n$ the set of all possible $(BF)^n$ paths. Similar sets can be defined for the $(FB)^n$ paths.

In this algorithm, instead of considering the number of $(BF)^n$ paths that leave i, it considers the number of $(BF)^n$ neighbours of node i. The contribution of node j to the weight of node i depends on the distance of the node j from i. By adopting an exponentially decreasing weighting scheme, the weight of node i is determined as:

$$a_i = 2^{n-1}|B(i)| + 2^{n-2}|BF(i)| + 2^{n-3}|BFB(i)| + \cdots + |(BF)^n(i)| \quad (13.30)$$

The algorithm starts from node i, and visits its neighbours in BFS order. At each iteration it takes a Backward or a Forward step (depending on whether it is an odd, or an even iteration), and it includes the new nodes it encounters. The weight factors are updated accordingly. Note that each node is considered only once, when it is first encountered by the algorithm.

13.1.4.7 Bayesian Algorithm

Reference [6] also proposed a fully Bayesian statistical approach to authorities and hubs. If there are M hubs and N authorities, we suppose that each hub i has a

real parameter e_i, corresponding to its tendency to have hypertext links, and also a non-negative parameter h_i, corresponding to its tendency to have intelligent hypertext links to authoritative sites. Each authority j has a non-negative parameter a_j, corresponding to its level of authority.

The statistical model is as follows. The a priori probability of a link from hub i to authority j is given by

$$P(i \rightarrow j) = \frac{exp(a_j h_i + e_i)}{1 + exp(a_j h_i + e_i)} \tag{13.31}$$

with the probability of no link from i to j given by

$$P(i \not\rightarrow j) = \frac{1}{1 + exp(a_j h_i + e_i)} \tag{13.32}$$

This states that a link is more likely if e_i is large (in which case hub i has large tendency to link to any site), or if both h_i and a_j are large (in which case i is an intelligent hub, and j is a high-quality authority).

We must now assign prior distributions to the $2M + N$ unknown parameters e_i, h_i, and a_j. To do this, we let $\mu = -5.0$ and $\sigma = 0.1$ be fixed parameters, and let each e_i have prior distribution $N(\mu, \sigma^2)$, a normal distribution with mean μ and variance σ^2. We further let each h_i and a_j have prior distribution $exp(1)$, meaning that for $x \geq 0$, $P(h_i \geq x) = P(a_j \geq x) = exp(-x)$. The Bayesian inference method then proceeds from this fully-specified statistical model, by conditioning on the observed data, which in this case is the matrix A of actual observed hypertext links in the Base Set. Specifically, when we condition on the data A we obtain a posterior density $\prod : R^{2M+N} \rightarrow [0, \infty)$ for the parameters $(e_1, \ldots, e_M, h_1, \ldots, h_M, a_1, \ldots, a_N)$. This density is defined so that

$$\begin{aligned} &P((e_1, \ldots, e_M, h_1, \ldots, h_M, a_1, \ldots, a_N) \in S | \{A_{ij}\}) \\ &\quad = \int_S \pi(e_1, \ldots, e_M, h_1, \ldots, h_M, a_1, \ldots, a_N) \\ &\qquad de_1 \ldots de_M dh_1 \ldots dh_M da_1 \ldots da_N \end{aligned} \tag{13.33}$$

for any measurable subset $S \subseteq R^{2M+N}$, and also

$$\begin{aligned} &E(g(e_1, \ldots, e_M, h_1, \ldots, h_M, a_1, \ldots, a_N) | \{A_{ij}\}) \\ &= \int_{R^{2M+N}} g(e_1, \ldots, e_M, h_1, \ldots, h_M, a_1, \ldots, a_N) \\ &\qquad \pi(e_1, \ldots, e_M, h_1, \ldots, h_M, a_1, \ldots, a_N) \\ &\qquad de_1 \ldots de_M dh_1 \ldots dh_M da_1 \ldots da_N \end{aligned} \tag{13.34}$$

for any measurable function $g : R^{2M+N} \rightarrow R$.

The posterior density for the model is given up to a multiplicative constant, by

$$\pi(e_1,\ldots,e_M,h_1,\ldots,h_M,a_1,\ldots,a_N) \propto \prod_{i=0}^{M-1} exp(-h_i)exp[-(e_i-\mu)^2/(2\sigma^2)]$$
$$\times \prod_{j=0}^{N-1} exp(-a_j) \times \prod_{(i,j):A_{ij}=1} exp(a_j h_i + e_i) / \prod_{all\ i,j} (1 + exp(a_j h_i + e_i)) \tag{13.35}$$

The Bayesian algorithm then reports the conditional means of the $2M + N$ parameters, according to the posterior density π, i.e, it reports final values $\hat{a_j}$, $\hat{h_i}$, and $\hat{e_i}$, where, for example

$$\hat{a_j} = \int_{R^{2M+N}} a_j \pi(e_1,\ldots,e_M,h_1,\ldots,h_M,a_1,\ldots,a_N) \, de_1 \ldots de_M dh_1 \ldots dh_M da_1 \ldots da_N \tag{13.36}$$

A Metropolis algorithm is used to compute these conditional means.

This algorithm can be further simplified by replacing Eq. 13.31 with $P(i \rightarrow j) = (a_j h_i)/(1 + a_j h_i)$ and replacing Eq. 13.32 with $P(i \not\rightarrow j) = 1/(1 + a_j h_i)$. This eliminates the parameters e_i entirely, so that prior values μ and σ are no longer needed. This leads to the posterior density $\pi(\cdot)$, now given by $\pi : R^{M+N} \rightarrow R^{\geq 0}$ where

$$\pi(h_1,\ldots,h_M,a_1,\ldots,a_N) \propto \prod_{i=0}^{M-1} exp(-h_i) \times \prod_{j=0}^{N-1} exp(-a_j) \times \prod_{(i,j):A_{ij}=1} a_j h_i \Big/ \prod_{all\ i,j} (1 + a_j h_i) \tag{13.37}$$

Comparison of these algorithms can be found in [6].

13.2 Google

Reference [7] describes *Google*, a prototype of a large-scale search engine which makes use of the structure present in hypertext. This prototype forms the base of the Google search engine we know today.

Figure 13.10 illustrates a high level view of how the whole system works. The paper states that most of Google was implemented in C or C++ for efficiency and was available to be run on either Solaris or Linux.

The *URL Server* plays the role of the crawl control module here by sending the list of URLs to be fetched by the crawlers. The *Store Server* receives the fetched pages and compresses them before storing it in the repository. Every Webpage has an associated ID number called a docID which is assigned whenever a new URL is parsed out of a Webpage. The *Indexer* module reads the repository, uncompresses the

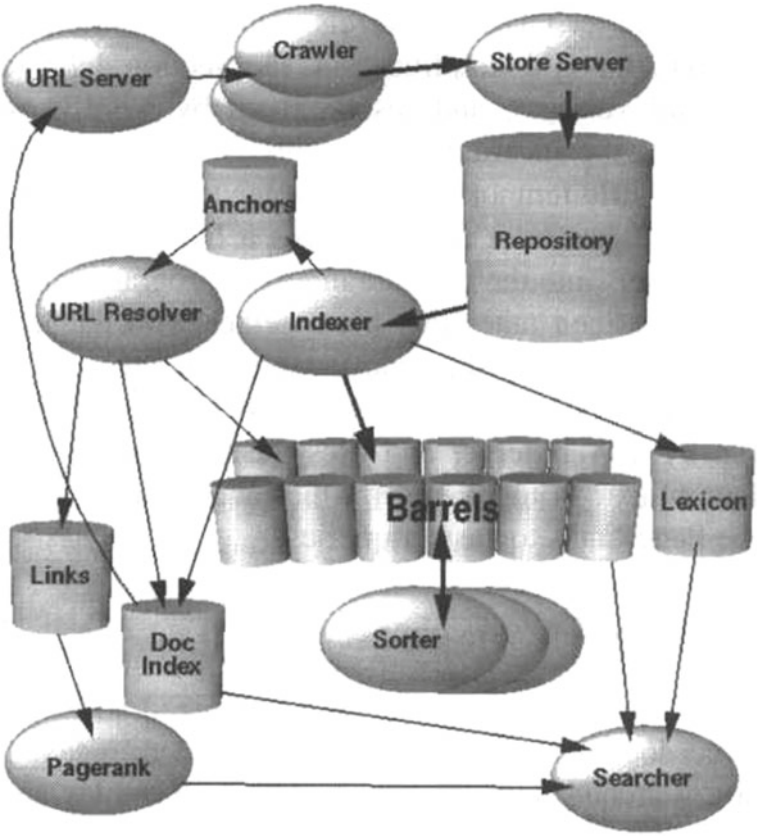

Fig. 13.10 High level Google architecture

documents and parses them. Each of these documents is converted to a set of word occurrences called hits. The hit record the word, position in document, approximation of the font size and capitalization. These hits are then distributed into a set of *Barrels*, creating a partially sorted forward index. The indexer also parses out all the links in every Webpage and stores important information about them in anchor files placed in *Anchors*. These anchor files contain enough information to determine where each link points from and to, and the text of the link.

The *URL Resolver* reads these anchor files and converts relative URLs into absolute URLs and in turn into docIDs. These anchor texts are put into the forward index, associated with the docID that the anchor points to. It also generates a database of links which are pairs of docIDs. The *Links* database is used to compute PageRanks for all the documents. The *Sorter* takes the barrels, which are sorted by docID, and resorts them by wordID to generate the inverted index. The sorter also produces a list of wordIDs and offsets into the inverted index. A program called *DumpLexicon* takes this list together with the lexicon produced by the indexer and generates a new

lexicon to be used by the searcher. The *Searcher* is run by a Web server and uses the lexicon built by DumpLexicon together with the inverted index and the *PageRanks* to answer queries.

13.2.1 Data Structures

Almost all of the data is stored in BigFiles which are virtual files that can span multiple file systems and support compression. The raw HTML repository uses roughly half of the necessary storage. It consists of concatenation of the compressed HTML of every page, preceded by a small header. The *DocIndex* keeps information about each document. The DocIndex is a fixed width Index Sequential Access Mode (ISAM) index, ordered by docID. The information stored in each entry includes the current document status, a pointer into the repository, a document checksum, and various statistics. Variable width information such as URL and title is kept in a separate file. There is also an auxiliary index to convert URLs into docIDs. The lexicon has several different forms for different operations. They all are memory-based hash tables with varying values attached to each word.

Hit lists account for most of the space used in both the forward and the inverted indices. Because of this, it is important to represent them as efficiently as possible. Therefore, a hand optimized compact encoding is used which uses two bytes for every hit. To save space, the length of the hit list is stored before the hits and is combined with the wordID in the forward index and the docID in the inverted index.

The forward index is actually already partially sorted. It is stored in a number of barrels. Each barrel holds a range of wordIDs. If a document contains words that fall into a particular barrel, the docID is recorded into the barrel, followed by a list of wordIDs with hitlists which correspond to those words. This scheme requires slightly more storage because of duplicated docIDs but the difference is very small for a reasonable number of buckets and saves considerable time and coding complexity in the final indexing phase done by the sorter. The inverted index consists of the same barrels as the forward index. Except that they have been processed by the sorter. For every valid wordID, the lexicon contains a pointer into the barrel that wordID falls into. It points to a list of docIDs together with their corresponding hit lists. This list is called a doclist and represents all the occurrences of that word in all documents.

An important issue is in what order the docIDs should appear in the doclist. One simple solution is to store them sorted by docID. This allows for quick merging of different doclists for multiple word queries. Another option is to store them sorted by a ranking of the occurrence of the word in each document. This makes answering one word queries trivial and makes it likely that the answers to multiple word queries are near the start. However, merging is much more difficult. Also, this makes development much more difficult in that a change to the ranking function requires a rebuild of the index. A compromise between these options was chosen, keeping two sets of inverted barrels - one set for hit lists which include title or anchor hits and

another set for all hit lists. This way, we check the first set of barrels first and if there are not enough matches within those barrels we check the larger ones.

13.2.2 Crawling

Crawling is the most fragile application since it involves interacting with hundreds of thousands of Web servers and various name servers which are all beyond the control of the system. In order to scale to hundreds of millions of Webpages, Google has a fast distributed crawling system. A single URLserver serves lists of URLs to a number of crawlers. Both the URLserver and the crawlers were implemented in Python. Each crawler keeps roughly 300 connections open at once. This is necessary to retrieve Web pages at a fast enough pace. At peak speeds, the system can crawl over 100 Web pages per second using four crawlers. A major performance stress is DNS lookup so each crawler maintains a DNS cache. Each of the hundreds of connections can be in a number of different states: looking up DNS, connecting to host, sending request. and receiving response. These factors make the crawler a complex component of the system. It uses asynchronous IO to manage events, and a number of queues to move page fetches from state to state.

13.2.3 Searching

Every hitlist includes position, font, and capitalization information. Additionally, hits from anchor text and the PageRank of the document are factored in. Combining all of this information into a rank is difficult. The ranking function so that no one factor can have too much influence. For every matching document we compute counts of hits of different types at different proximity levels. These counts are then run through a series of lookup tables and eventually are transformed into a rank. This process involves many tunable parameters.

13.3 Web Spam Pages

Web spam pages are notorious for using techniques to achieve higher-than-deserved rankings in a search engine's results. Reference [12] proposed a technique to semi-automatically separate reputable pages from spam. First, a small set of seed pages are evaluated to be good by a human expert. Once these reputable seed pages have been identified, the link structure of the Web is used to discover other pages that are likely to be good.

Humans can easily identify spam pages and a number of search engine companies have been known to recruit people who have expertise in identifying spam, and

regularly scan the Web looking for such pages. However, this is an expensive and time-consuming process but nevertheless has to be done in order to ensure the quality of the search engine's result.

The algorithm first selects a small seed set of pages whose spam status needs to be determined by a human. Using this, the algorithm identifies other pages that are likely to be good based on their connectivity with the good seed pages.

The human checking of a page for spam is formalized as a *oracle function* O over all pages $p \in V$

$$O(p) = \begin{cases} 0 \; if \;\; p \; is \; bad \\ 1 \; if \;\; p \; is \; good \end{cases} \tag{13.38}$$

Since oracle evaluations over all pages is an expensive ordeal, this is avoided by asking the human to assign oracle values for just a few of the pages.

To evaluate pages without relying on O, the likelihood that a given page p is good is estimated using a *trust function* T which yields the probability that a page is good, i.e, $T(p) = P[O(p) = 1]$.

Although it would be difficult to come up with a function T, such a function would be useful in ordering results. These functions could be defined in terms of the following properties. We first look at the *ordered trust property*

$$\begin{aligned} T(p) < T(q) &\Leftrightarrow P[O(p) = 1] < P[O(q) = 1] \\ T(p) = T(q) &\Leftrightarrow P[O(p) = 1] = P[O(q) = 1] \end{aligned} \tag{13.39}$$

By introducing a threshold value δ, we define the *threshold trust property* as

$$T(p) > \delta \Leftrightarrow O(p) = 1 \tag{13.40}$$

A binary function $I(T, O, p, q)$ to signal if the ordered trust property has been violated.

$$I(T, O, p, q) = \begin{cases} 1 \; if \;\; T(p) \geq T(q) \; and \; O(p) < O(q) \\ 1 \; if \;\; T(p) \leq T(q) \; and \; O(p) > O(q) \\ 0 \; otherwise \end{cases} \tag{13.41}$$

We now define a set of $\mathcal{P}$ of ordered pairs of pages (p, q), $p \neq q$, and come up with the *pairwise orderedness* function that computes the fraction of pairs for which T did not make a mistake

$$pairord(T, O, \mathcal{P}) = \frac{|\mathcal{P}| - \sum\limits_{(p,q) \in \mathcal{P}} I(T, O, p, q)}{|\mathcal{P}|} \tag{13.42}$$

If $pairord$ equals 1, there are no cases when misrated a pair. In contrast, if $pairord$ equals 0, then T misrated all the pairs.

We now proceed to define the trust functions. Given a budget L of O-invocations, we select at random a seed set $\mathcal{S}$ of L pages and call the oracle on its elements. The subset of good and bad seed pages are denoted as $\mathcal{S}^+$ and $\mathcal{S}^-$, respectively. Since the remaining pages are not checked by human expert, these are assigned a trust score of $1/2$ to signal our lack of information. Therefore, this scheme is called the *ignorant trust function* T_0 defined for any $p \in V$ as

$$T_0(p) = \begin{cases} O(p) \; if \; p \in \mathcal{S} \\ 1/2 \; otherwise \end{cases} \tag{13.43}$$

Now we attempt to compute the trust scores, by taking advantage of the approximate isolation of good pages. We will select at random the set $\mathcal{S}$ of L pages that we invoke the oracle on. Then, expecting that good pages point to other good pages only, we assign a score of 1 to all pages that are reachable from a page in $\mathcal{S}^+$ in M or fewer steps. The *M-step trust function* is defined as

$$T_M(p) = \begin{cases} O(p) \; if \; p \in \mathcal{S} \\ 1 \; if \; p \notin \mathcal{S} \; and \; \exists q \in \mathcal{S}^+ : q \rightsquigarrow_M p \\ 1/2 \; otherwise \end{cases} \tag{13.44}$$

where $q \rightsquigarrow_M p$ denotes the existence of a path of maximum length of M from page q to page p. However, such a path must not include bad seeds.

To reduce trust as we move further away from the good seed pages, there are two possible schemes. The first is called the *trust dampening*. If a page is one link away from a good seed page, it is assigned a dampened trust score of β. A page that can reach another page with score β, gets a dampened score of $\beta \cdot \beta$.

The second technique is called *trust splitting*. Here, the trust gets split if it propagates to other pages. If a page p has a trust score of $T(p)$ and it points to $\omega(p)$ pages, each of the $\omega(p)$ pages will receive a score fraction $T(p)/\omega(p)$ from p. In this case, the actual score of a page will be the sum of the score fractions received through its in-links.

The next task is to identify pages that are desirable for the seed set. By desirable, we mean pages that will be the most useful in identifying additional good pages. However, we must ensure that the size of the seed set is reasonably small to limit the number of oracle invocations. There are two strategies for accomplishing this task.

The first technique is based on the idea that since trust flows out of the good seed pages, we give preference to pages from which we can reach many other pages. Building on this, we see that seed set can be built from those pages that point to many pages that in turn point to many pages and so on. This approach is the inverse of the PageRank algorithm because the PageRank ranks pages based on its in-degree while here it is based in the out-degree. This gives the technique the name *inverted PageRank*. However, this method is a double-edged sword. While it does not guarantee maximum coverage, its execution time is polynomial in the number of pages.

The other technique is to take pages with high PageRank as the seed set, since high-PageRank pages are likely to point to other high- PageRank pages.

Piecing all of these elements together gives us the *TrustRank* algorithm. The algorithm takes as input the graph. At the first step, it identifies the seed set. The pages in the set are re-ordered in decreasing order of their desirability score. Then, the oracle function is invoked on the L most desirable seed pages. The entries of the static score distribution vector d that correspond to good seed pages are set to 1. After normalizing the vector d so that its entries sum to 1, the TrustRank scores are evaluated using a biased PageRank computation with d replacing the uniform distribution.

Reference [13] complements [12] by proposing a novel method for identifying the largest *spam farms*. A spam farm is a group of interconnected nodes involved in link spamming. It has a single *target node*, whose ranking the spammer intends to boost by creating the whole structure. A farm also contains *boosting nodes*, controlled by the spammer and connected so that they would influence the PageRank of the target. Boosting nodes are owned either by the author of the target, or by some other spammer. Commonly, boosting nodes have little value by themselves; they only exist to improve the ranking of the target. Their PageRank tends to be small, so serious spammers employ a large number of boosting nodes (thousands of them) to trigger high target ranking.

In addition to the links within the farm, spammers may gather some external links from reputable nodes. While the author of a reputable node y is not voluntarily involved in spamming, "stray" links may exist for a number of reasons. A spammer may manage to post a comment that includes a spam link in a reputable blog. Or a *honey pot* may be created, a spam page that offers valuable information, but behind the scenes is still part of the farm. Unassuming users might then point to the honey pot, without realizing that their link is harvested for spamming purposes. The spammer may purchase domain names that recently expired but had previously been reputable and popular. This way he/she can profit of the old links that are still out there.

A term called *spam mass* is introduced which is a measure of how much PageRank a page accumulates through being linked to by spam pages. The idea is that the target pages of spam farms, whose PageRank is boosted by many spam pages, are expected to have a large spam mass. At the same time, popular reputable pages, which have high PageRank because other reputable pages point to them, have a small spam mass.

In order to compute this spam mass, we partition the Web into a set of reputable node V^+ and a set of spam nodes V^-, with $V^+ \cup V^- = V$ and $V^+ \cap V^- = \phi$. Given this partitioning, the goal is to detect web nodes x that gain most of their PageRank through spam nodes in V^- that link to them. Such nodes x are called spam farm target nodes.

A simple approach would be that, given a node x, we look only at its immediate in-neighbours. If we are provided information about whether or not these in-neighbours are reputable or spam, we could infer whether or not x is good or spam.

In a first approach, if majority of x's links comes from spam nodes, x is labelled a spam target node, otherwise it is labelled good. This scheme could easily mislabel spam. An alternative is to look not only at the number of links, but also at what

amount of PageRank each link contributes. The contribution of a link amounts to the change in PageRank induced by the removal of the link. However, this scheme does not look beyond the immediate in-neighbours of x and therefore could succumb to mislabelling. This paves a way for a third scheme where a node x is labelled considering all the PageRank contributions of other nodes that are directly or indirectly connected to x.

The *PageRank contribution* of x to y over the *walk* W is defined as

$$q_y^W = s^k \pi(W)(1-s)v_x \tag{13.45}$$

where v_x is the probability of a random jump to x, and $\pi(W)$ is the weight of the walk.

$$\pi(W) = \prod_{i=0}^{k-1} \frac{1}{out(x_i)} \tag{13.46}$$

This weight can be interpreted as the probability that a Markov chain of length k starting in x reaches y through the sequence of nodes $x_1, \ldots, x_{k-1}$.

This gives the total PageRank contribution of x to y, $x \neq y$, over all walks from x to y as

$$q_y^x = \sum_{W \in W_{xy}} q_y^W \tag{13.47}$$

For a node's contribution to itself, a virtual cycle Z_x that has length zero and weight 1 is considered such that

$$\begin{aligned} q_x^x = \sum_{W \in W_{xx}} q_x^W &= q_x^{Z_x} + \sum_{V \in W_{xx}, |V| \geq 1} q_x^V \\ &= (1-s)v_x + \sum_{V \in W_{xx}, |V| \geq 1} q_x^V \end{aligned} \tag{13.48}$$

If a node x does not participate in cycles, x's contribution to itself is $q_x^x = (1-s)v_x$, which corresponds to the random jump component. If these is no walk from node x to y then the PageRank contribution q_y^x is zero.

This gives us that the PageRank *score* of a node y is the sum of the contributions of all other nodes to y

$$p_y = \sum_{x \in V} q_y^x \tag{13.49}$$

Under a given random jump distribution v, the vector q^x of contributions of a node x to all nodes is the solution of the linear PageRank system for the core-based random jump vector v^x

$$v_y^x = \begin{cases} v_x \; if \; x = y \\ 0 \; otherwise \end{cases} \implies q^x = PR(v^x) \tag{13.50}$$

From this, we can compute the PageRank contribution q^U of any subset of nodes $U \subseteq V$ by computing the PageRank using the random jump vector v^U defined as

$$v_y^U = \begin{cases} v_y \; if \; y \in U \\ 0 \; otherwise \end{cases} \tag{13.51}$$

We will now use this to compute the spam mass of a page. For a given partitioning $\{V^+, V^-\}$ of V and for any node x, $p_x = q_x^{V^+} + q_x^{V^-}$, i.e, x's PageRank is the sum of the contributions of good and that of spam nodes. The *absolute spam mass* of a node x, denoted by M_x, is the PageRank contribution that x receives from spam nodes, i.e, $M_x = q_x^{V^-}$. Therefore, the spam mass is a measure of how much direct or indirect in-neighbour spam nodes increase the PageRank of a node. The *relative spam mass* of node x, denoted by m_x, is the fraction of x's PageRank due to contributing spam nodes, i.e, $m_x = q_x^{V^-}/p_x$.

If we assume that only a subset of the good nodes $\tilde{V^+}$ is given, we can compute two sets of PageRanks scores. $p = PR(v)$ is the PageRank of nodes based on the uniform random jump distribution $v = (1/n)^n$, and $p' = PR(v^{\tilde{V^+}})$ is a core-based PageRank with a random jump distribution $v^{\tilde{V^+}}$

$$v_x^{\tilde{V^+}} = \begin{cases} 1/n \; if \; x \in \tilde{V^+} \\ 0 \; otherwise \end{cases} \tag{13.52}$$

Given the PageRank scores p_x and p'_x, the estimated absolute spam mass of node x is

$$\tilde{M}_x = p_x - p'_x \tag{13.53}$$

and the estimated spam mass of x is

$$\tilde{m_x} = (p_x - p'_x)/p_x = 1 - p'_x/p_x \tag{13.54}$$

Instead, if $\tilde{V^-}$ is provided, the absolute spam mass can be estimated by $\hat{M} = PR(v^{\tilde{V^-}})$. When both V^- and V^+ are known, the spam mass estimates could be derived by simply computing the average $(\tilde{M} + \hat{M})/2$.

Now, we consider the situation where the core $\tilde{V^+}$ is significantly smaller than the actual set of good nodes V^+, i.e, $|\tilde{V^+}| << |V^+|$ and thus $||v^{V^+}|| << ||v||$. Since $p = PR(v)$, $||p|| \leq ||v||$ and similarly $||p'|| \leq ||v^{\tilde{V^+}}||$. This means that $||p'|| << ||p||$, i.e, the total estimated good contribution is much smaller than the

total PageRank of nodes. Therefore, we will have $||p - p'|| \approx ||p||$ with only a few nodes that have absolute mass estimates differing from their PageRank scores.

To counter this problem, we construct a uniform random sample of nodes and manually label each sample node as spam or good. This way it is possible to roughly approximate the prevalence of spam nodes on the Web. We introduce γ to denote the fraction of nodes that we estimate that are good, so $\gamma n \approx |V^+|$. Then, we scale the core-based random jump vector $v^{\tilde{V^+}}$ to w, where

$$w_x = \begin{cases} \gamma/|\tilde{V^+}| \; if \; x \in \tilde{V^+} \\ 0 \; otherwise \end{cases} \tag{13.55}$$

Note that $||w|| = \gamma \approx ||v^{V^+}||$, so the two random jump vectors are of the same order of magnitude. Then, we can compute p' based on w and expect that $||p'|| \approx ||p^{V^+}||$, so we get a reasonable estimate of the total good contribution.

Using w in computing the core-based PageRank leads to the following: as $\tilde{V^+}$ is small, the good nodes in it will receive an unusually high random jump ($\gamma/|\tilde{V^+}|$ as opposed to $1/n$). Therefore, the good PageRank contribution of these known reputable nodes will be overestimated, to the extent that occasionally for some node y, p'_y will be larger than p_y. Hence, when computing $\tilde{M}$, there will be nodes with *negative spam mass*. In general, a negative mass indicates that a node is known to be good in advance (is a member of $\tilde{V^+}$) or its PageRank is heavily influenced by the contribution of nodes in the good core.

Now we can discuss the mass-based spam detection algorithm. It takes as input a good core $\tilde{V^+}$, a threshold τ to which the relative mass estimates are compared, and a PageRank threshold ρ. If the estimated relative mass of a node is equal to or above this τ then the node is labelled as a spam candidate. Only the relative mass estimates of nodes with PageRank scores larger than or equal to ρ are verified. Nodes with PageRank less than ρ are never labelled as spam candidates.

Problems

Download the Wikipedia hyperlinks network available at https://snap.stanford.edu/data/wiki-topcats.txt.gz.

63 Generate the graph of the network in this dataset.

64 Compute the PageRank, hub score and authority score for each of the nodes in the graph.

65 Report the nodes that have the top 3 PageRank, hub and authority scores respectively.

References

1. Agirre, Eneko, Mona Diab, Daniel Cer, and Aitor Gonzalez-Agirre. 2012. Semeval-2012 task 6: a pilot on semantic textual similarity. In *Proceedings of the first joint conference on lexical and computational semantics-volume 1: Proceedings of the main conference and the shared task, and Volume 2: Proceedings of the sixth international workshop on semantic evaluation*, 385–393. Association for Computational Linguistics.
2. Altman, Alon, and Moshe Tennenholtz. 2005. Ranking systems: the pagerank axioms. In *Proceedings of the 6th ACM conference on electronic commerce*, 1–8. ACM.
3. Arasu, Arvind, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. 2001. Searching the web. *ACM Transactions on Internet Technology (TOIT)* 1 (1): 2–43.
4. Berkhin, Pavel. 2005. A survey on pagerank computing. *Internet Mathematics* 2 (1): 73–120.
5. Bharat, Krishna, and Monika R. Henzinger. 1998. Improved algorithms for topic distillation in a hyperlinked environment. In *Proceedings of the 21st annual international ACM SIGIR conference on research and development in information retrieval*, 104–111. ACM.
6. Borodin, Allan, Gareth O. Roberts, Jeffrey S. Rosenthal, and Panayiotis Tsaparas. 2001. Finding authorities and hubs from link structures on the world wide web. In *Proceedings of the 10th international conference on World Wide Web*, 415–429. ACM.
7. Brin, Sergey, and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems* 30 (1–7): 107–117.
8. Chakrabarti, Soumen, Byron Dom, Prabhakar Raghavan, Sridhar Rajagopalan, David Gibson, and Jon Kleinberg. 1998. Automatic resource compilation by analyzing hyperlink structure and associated text. *Computer Networks and ISDN Systems* 30 (1–7): 65–74.
9. Cho, Junghoo, and Hector Garcia-Molina. 2003. Estimating frequency of change. *ACM Transactions on Internet Technology (TOIT)* 3 (3): 256–290.
10. Cohn, David, and Huan Chang. 2000. Learning to probabilistically identify authoritative documents. In *ICML*, 167–174. Citeseer.
11. Dempster, Arthur P., Nan M. Laird, and Donald B. Rubin. 1977. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (methodological)* 1–38.
12. Gyöngyi, Zoltán, Hector Garcia-Molina, and Jan Pedersen. 2004. Combating web spam with trustrank. In *Proceedings of the thirtieth international conference on very large data bases-volume 30*, 576–587. VLDB Endowment.
13. Gyongyi, Zoltan, Pavel Berkhin, Hector Garcia-Molina, and Jan Pedersen. 2006. Link spam detection based on mass estimation. In *Proceedings of the 32nd international conference on very large data bases*, 439–450. VLDB Endowment.
14. Haveliwala, Taher H. 2002. Topic-sensitive pagerank. In *Proceedings of the 11th international conference on World Wide Web*, 517–526. ACM.
15. Hirai, Jun, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. 2000. Webbase: a repository of web pages. *Computer Networks* 33 (1–6): 277–293.
16. Kleinberg, Jon M. 1998. Authoritative sources in a hyperlinked environment. In *In Proceedings of the ACM-SIAM symposium on discrete algorithms*, Citeseer.
17. Lempel, Ronny, and Shlomo Moran. 2000. The stochastic approach for link-structure analysis (salsa) and the tkc effect1. *Computer Networks* 33 (1–6): 387–401.
18. MacCluer, Charles R. 2000. The many proofs and applications of perron's theorem. *Siam Review* 42 (3): 487–498.
19. Melink, Sergey, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. 2001. Building a distributed full-text index for the web. *ACM Transactions on Information Systems (TOIS)* 19 (3): 217–241.
20. Rafiei, Davood, and Alberto O. Mendelzon. 2000. What is this page known for? computing web page reputations. *Computer Networks* 33 (1–6): 823–835.

21. Ribeiro-Neto, Berthier A., and Ramurti A. Barbosa. 1998. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the third ACM conference on digital libraries*, 182–190. ACM.
22. Tomasic, Anthony, and Hector Garcia-Molina. 1993. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the second international conference on parallel and distributed information systems*, 8–17. IEEE.